Journal of Discrete Algorithms 9 (2011) 100–121

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

# Position heaps: A simple and dynamic text indexing data structure

Andrzej Ehrenfeucht [a], Ross M. McConnell [b,*], Nissa Osheim [b], Sung-Whan Woo [b]

[a] Dept. of Computer Science, 430 UCB, University of Colorado at Boulder, Boulder, CO 80309-0430, USA
[b] Dept. of Computer Science, Colorado State University, Fort Collins, CO 80523-1873, USA

ARTICLE INFO

*Article history:*
Available online 9 December 2010

*Keywords:*
Position heap
String searching

ABSTRACT

We address the problem of finding the locations of all instances of a string $P$ in a text $T$, where preprocessing of $T$ is allowed in order to facilitate the queries. Previous data structures for this problem include the suffix tree, the suffix array, and the compact DAWG. We modify a data structure called a **sequence tree**, which was proposed by Coffman and Eve (1970) [3] for hashing, and adapt it to the new problem. We can then produce a list of $k$ occurrences of any string $P$ in $T$ in $O(\|P\| + k)$ time. Because of properties shared by suffixes of a text that are not shared by arbitrary hash keys, we can build the structure in $O(\|T\|)$ time, which is much faster than Coffman and Eve's algorithm. These bounds are as good as those for the suffix tree, suffix array, and the compact DAWG. The advantages are the elementary nature of some of the algorithms for constructing and using the data structure and the asymptotic bounds we can give for updating the data structure when the text is edited.



## 1. Introduction

In this paper, we consider the problem of finding occurrences of a pattern string $P$ in a text $T$, where preprocessing of $T$ is allowed in order to create a data structure that speeds up the search.

In this paper, we let $m$ denote the length $\|P\|$ of $P$, $n$ denote the length $\|T\|$ of $T$, and $k$ denote the number of positions in $T$ where $P$ occurs as a substring. We assume that the size of the alphabet $\Sigma$ is fixed.

We describe two data structures, the **position heap** and the **augmented position heap**. We gave a primitive version of the position heap in [5], and some of the results of this paper were sketched in [6], where we described a structure that is closely related to the augmented position heap, a *contracted suffix tree*. The position heap and the augmented position heap have helpful combinatorial properties that the contracted suffix tree does not. The position heap of $T$ is unique, while a contracted suffix tree is not.

**Definition 1.1.** Let $h(T)$ be the length of the longest substring $X$ of $T$ that is repeated at least $\|X\|$ times in $T$.

A few moment's reflection reveals that $h(T)$ can be expected to be quite small for most practical applications. The expected value of $h(T)$ is $O(\log n)$ when $T$ is a randomly-generated string. However, since few applications deal with random strings, a more important observation is that long repeated substrings in $T$ have little impact on the value of $h(T)$ unless they are repeated an inordinate number of times. We discuss properties of $h(T)$ in greater detail below.

In this paper, we give the following results:

* Corresponding author.
*E-mail addresses:* andrzej@cs.colorado.edu (A. Ehrenfeucht), rmm@cs.colostate.edu (R.M. McConnell), osheim@cs.colostate.edu (N. Osheim), woo@cs.colostate.edu (S.-W. Woo).

1570-8667/$ – see front matter 
doi:10.1016/j.jda.2010.12.001

1. We describe the augmented position heap for the first time. The data structure is a trie with $n$ nodes and height $O(h(T))$. It is augmented with some additional pointers to facilitate queries.
2. As a starting point for algorithms on the augmented position heap we review extremely simple algorithms for constructing the position heap in $O(nh(T))$ time, and for querying it in $O(\min(m^2, mh(T)))$ time [5]. Though these worst-case bounds are inferior to bounds we give below, the $O(\min(m^2, mh(T)))$ bound for the query algorithm is overly pessimistic in practice. For instance, when $T$ is a randomly constructed string and the construction of $P$ does not depend on $T$, or if $P$ is a randomly constructed string and construction of $T$ does not depend on $P$, then this simple query algorithm takes $O(m+k)$ expected time. Because of the simplicity of these algorithms and the expectation that $h(T)$ is usually small in practice, they are probably of practical interest in some contexts, and they are of pedagogical interest, as they can be taught and implemented in undergraduate data structures courses. In [5], we also gave a more sophisticated $O(n)$ algorithm for constructing the position heap that we generalize to the augmented position heap here.
3. We show how to get an $O(n)$ bound for constructing the augmented position heap and a simple $O(m+k)$ bound for finding the $k$ occurrences of $P$ in $T$. For the case where the user may wish to have the option to abandon the query unexpectedly after $k' < k$ occurrences have been returned, we show how to construct an iterator in $O(m)$ time that then gives occurrences of $P$ in $T$ in left-to-right order of occurrence in $O(\log k')$ time apiece.
4. We show that if one is willing to accept $O(|\Sigma|)$ instead of $O(\log|\Sigma|)$ for the implicit alphabet-size factor in the time bounds, the position heap takes $2n$ integers to represent. No additional space is required to construct it in $O(nh(T))$ time, and an additional $n$ integers are required during construction in order to construct it in $O(n)$ time. This implementation is attractive if the alphabet size is small.
5. We show how to adapt the position heap and the dynamic position heap for dynamically changing texts. When a consecutive block of $b$ characters is deleted from $T$, we show how to update the augmented position heap in $O((h(T)+b)h(T)\log n)$ amortized time. When a consecutive block of $b$ characters from is inserted to $T$, yielding text $T'$, we show how to update the augmented position heap in $O((h(T')+b)h(T')\log n)$ amortized time. The operations are based on the sift-up and sift-down operations on standard heaps. The reason for the $\log n$ factor and the amortization of the time bound is due to the data structure we use for maintaining the dynamic text, not for updating the augmented position heap.
   The tradeoff of implementing the position heap to accommodate string edits is that searches take $O(m\log n+k)$ amortized time, rather than $O(m+k)$ time.

Previous data structures for this problem include the *suffix tree* [13], the *compact directed acyclic word graph* (*compact DAWG*) [1], and the *suffix array* [9]. The first two approaches take $O(n)$ time to build the data structure, and $O(m+k)$ time to find the $k$ positions where the pattern string occurs.

The suffix array can be constructed in $O(n)$ time, and takes $O(m+\log n)$ time to produce a pointer to a list of occurrences of $P$ in $T$. A slightly slower approach takes $O(m\log n)$ time, and this approach is of practical interest because of its simplicity. When the text has low entropy, the FM-index scheme allows searching on a version of the text that is compressed with the Burrows–Wheeler transform [2] with no significant slowdown in the query time [8].

The biggest disadvantage of our structure when compared with suffix arrays is its larger space requirement. Like the suffix tree and the compact DAWG, and unlike the suffix array, our bounds must be increased by a $\log|\Sigma|$ factor when the size of the alphabet, $\Sigma$, is introduced as a variable. This factor comes from the time required to find the child of a node on the child edge labeled by a given letter of $\Sigma$. This can be improved to $O(1)$ *expected* time with a hash table that returns the child, given a hash key consisting of the parent and a letter. This is nevertheless also a disadvantage when compared to suffix arrays.

There has been previous work on updating indexing structures when the text is edited. The *generalized suffix tree* allows a search for a pattern string in a collection of texts. In [7], it is shown that it is possible to implement it to allow insertion and removal of any text $X$ in the collection in $O(\|X\|)$ time, and in [8], it is shown how to do this on a collection of Burrows–Wheeler compressed texts in near-linear time. However, $X$ must be inserted or removed in its entirety and arbitrary edits on $X$ are not supported.

The results most comparable to ours have been given recently by Salson et al. [11,10]. They have given an approach that takes $O(n)$ worst-case time to modify the Burrows–Wheeler transform and the suffix array after an arbitrary edit operation on $T$ [11,10]. Though, in the worst case, this is as bad as the cost of discarding the suffix array and rebuilding it from the beginning, they argue that their approach is much more efficient in practice, and support this with empirical studies on benchmarks.

Note that $h(T)$ is also $\Theta(n)$ in the worst case, such as when $T = a^n b$. However, our bounds are stronger than $O(n)$ because we can always rebuild the data structure from scratch in $O(n)$ time if $O((h(T)+b)h(T)\log n)$ exceeds this cost, and it characterizes analytically the relationship between the running time and an easily-understood property of the text.

Salson et al. identify $a^n b$ as a text that requires $\Theta(n)$ time for their algorithm also. However, the performance of their algorithms can suffer greatly from a *single* repetition of a large string in $T$. An illustration of this phenomenon is where $W$ is a string on $\Sigma$, \$ is a special character they append to $T$ that is less than any letter in $\Sigma$ in lexical order, and $T$ is the concatenation $WW\$$. Suppose # is a character that is larger than any character in $\Sigma$ in lexical order. Let $T' = WW\#\$$. Then $\Theta(n)$ changes to the suffix array of $T$ are required to obtain the suffix array of $T'$. By contrast, $h(T) = O(h(W))$ in this case,

it takes $O((h(W)+b)h(W)\log n)$ amortized bound to update our data structure after this same edit operation. This is seen by the following lemma, which gives an overly pessimistic upper bound on $h(T)$ in terms of $h(W)$.

**Lemma 1.2.** *If $T = WW$, then $h(T) \leqslant 2h(W)$.*

**Proof.** Let $X$ be a string of length $h(T)$ that occurs $h(T)$ times in $T$. For each of these occurrences, either the first $\lceil h(T)/2 \rceil$ characters of the occurrence lie in the first occurrence of $W$, or the last $\lceil h(T)/2 \rceil$ characters lie in the second occurrence of $W$. By the pigeonhole principle, one of these is a string of length $\lceil h(T)/2 \rceil$ that occurs $\lceil h(T)/2 \rceil$ times in $W$, giving a lower bound of $\lceil h(T)/2 \rceil$ on $h(W)$. □

Implementations of algorithms and data structures given in this paper can be found at www.cs.colostate.edu/PositionHeaps.

## 2. Preliminaries

Let $\lambda$ be the null string. If $X = x_1 x_2 \dots x_j$ is a string, we let $\|X\|$ denote the length $j$ of $X$. The **reverse** of $X$ is the string $X^R = x_j x_{j-1} \dots x_1$.

For reasons that will become clear shortly, we adopt the convention of numbering the positions of the text $T$ from right to left, so $T = t_n t_{n-1} \dots t_1$. Let $T_i$ denote the suffix $t_i t_{i-1} \dots t_1$ beginning at position $i$. Let us distinguish a **substring** $P = p_1 p_2 \dots p_m$ of a $T$ from an **instance** $i$ of $P$ in $T$, where $P = t_i t_{i-1} \dots t_{i-m+1}$. The null substring, $\lambda$, is considered to occur at every position.

If $X$ and $Y$ are strings, we denote their concatenation by $XY$. If $X$ is a prefix of $P$, we let $P - X$ denote the suffix of $P$ consisting of the last $\|P\| - \|X\|$ letters of $P$. If $X$ is a suffix of $P$, we let $P \setminus X$ denote the prefix of $P$ consisting of the first $\|P\| - \|X\|$ letters of $P$.

**Definition 2.1.** A rooted tree has the **heap property** if each node carries a label from an ordered set, such as the integers, and, for every internal node $X$, the labels of the children of $X$ are greater than the label of $X$.

**Definition 2.2.** A **trie** on alphabet $\Sigma$ denotes a rooted tree $T$ with the following properties:

1. Each edge is labeled with a character;
2. For each node $u$ and letter $b \in \Sigma$, there is at most one edge with label $b$ from $u$ to a child of $u$.

Given a trie, let us say that the **label of a path** from the root to a node $u$ is the string given by the sequence $X$ of characters that occur on edges of the path. This is the **path label** of $u$. Because of the second property, the path label uniquely identifies $u$. We therefore adopt the convention of treating the node and its path label as interchangeable objects. For example, we may consider whether a *string* $X$ is a *node* of the trie, or whether one *node* is a *substring* of another. Note that one node is a prefix of another if and only if it is an ancestor in the trie.

A basic operation on a trie takes an input string $P = p_1 p_2 \dots p_m$ and finds the largest prefix $P'$ of $P$ that is a node of the trie. Since $|\Sigma|$ is fixed, this is easily accomplished in $O(\|P'\|)$ time by starting at the root and iteratively taking edges labeled with the sequence of letters from $P$, until $P$ is exhausted or a node is encountered that doesn't have a child on the next letter of $P$. Let us call this operation **indexing** into the trie.

## 3. Sequence hash trees

A data structure of Coffman and Eve [3], called a **sequence hash tree**, was designed for the problem of implementing hash tables (dictionaries) whose keys are strings. It consists of a trie for indexing into the table. The structure of the tree depends on the order in which the strings are inserted. We describe a minor variant that is easier to adapt to our substring matching problem, below.

Let $\mathcal{S} = (S_1, S_2, \dots, S_n)$ be a given ordering of the strings. Without loss of generality for our purposes, we may assume that no string in $\mathcal{S}$ is a prefix of any other. The trie $H_n$ that they construct is defined by induction, as follows. If $i = 1$, the trie $H_1$ is just a root node with a pointer to $S_1$. If $i > 1$, then $H_i$ is obtained from $H_{i-1}$ by finding the shortest prefix $Xb$ of $S_i$ that is not already a node of the trie. A new node $Xb$ is added as the child of node $X$ on edge labeled $b$, and a pointer is installed from it to $S_i$.

Fig. 1 gives an example. Coffman and Eve's paper has received little attention since it was published in 1970, due, in no doubt, to the existence of superior ways of implementing a hash table. In the present paper, we show that this data structure is much richer when considered in the context of the new problem. The structure of the set of suffixes of a text $T$ allows us to derive interesting and algorithmically useful properties that do not apply in the general case addressed by Coffman and Eve. In particular, we show that it has height at most $h(T)$, and show that if the suffixes are inserted in ascending order of length, it is now possible to build the data structure in time that is linear in $n = \|T\|$, that is, in $O(1)$ time, amortized, per hash key. We show how the tree can be augmented with *maximal-reach pointers* so that finding all $k$ entries that have $P$ as a prefix takes $O(m+k)$ worst-case time, independently of the height of the tree.

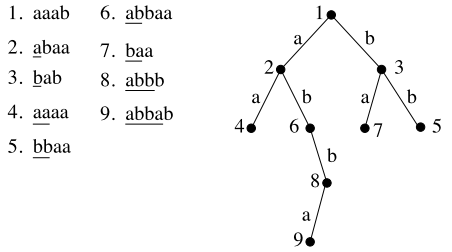


**Fig. 1.** The sequence hash tree of a sequence of strings. We refer to each node by the string of letter labels on the path from the root to the node. For example, the node labeled 6 can be thought of as synonymous with the string *ab*. Each string in the sequence is installed at a new node that is the shortest prefix of the string that isn't already a node of the sequence hash tree. These prefixes are underlined. For example, when string 9 is inserted, its prefix *abb* is already a node of the tree, but its prefix *abba* is not, so a pointer to string 9 is inserted at a new node, *abba*.

| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | a | a | b | a | b | b | a | b | b | a | b |

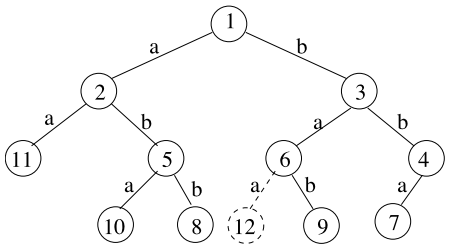

**Fig. 2.** Incremental construction of the position heap. Suffixes $T_1, T_2, \ldots, T_n$ are inserted in ascending order of length. The figure depicts the insertion of $T_i$ when $i = 12$. Indexing into the heap on $T_i$ identifies the longest prefix (*ba*) of $T_i$ that is already a node $Y$ of the heap. The shortest prefix of $T_i$ that is not already a node of the heap (*baa*) is inserted as a child of $Y$ and labeled with position $i$.

## 4. The position heap

Up until the last two sections of this paper, we assume that $T$ is static. We can therefore suppose that $T$ and $P$ are stored in character arrays, which supports lookup of the character in a given position $i$ in $O(1)$ time.

**Definition 4.1.** The **position heap** $H(T)$ of a text $T$ is obtained by iteratively inserting the suffixes $(T_1, T_2, \ldots, T_n)$ of $T$, in ascending order of length, into Coffman and Eve's data structure using their insertion operation. That is, $T_i$ is inserted by creating a new node that is the shortest prefix of $T_i$ that is not already a node of the tree, and labeling it with position $i$.

Let us call the algorithm implied by this constructive definition the **naive construction algorithm**. Fig. 2 gives an illustration. Coffman and Eve assume that each inserted string ends with a special character \$, because one must ensure that no inserted string is already a node of the tree when it is inserted. The use of a special character to accomplish this is unnecessary in our context, since each string $T_i$ is longer than any string inserted before it, and each node previously inserted is a prefix of some $T_j$ for $j < i$.

The construction can be executed for any text $T$, and, since it is deterministic, the position heap $H(T)$ for a text is unique. The algorithm is simple enough to be taught and programmed in undergraduate data structures classes.

### *4.1. A time bound for constructing the position heap*

We now give a time bound for using the above constructive definition of the position heap as an algorithm. We improve the time bound to $O(n)$ below, at the expense of adding elements to the data structure.

**Lemma 4.2.** *The height of the position heap of a text $T$ is at most $2h(T)$.*

**Proof.** Let $X = x_j x_{j-1} \ldots x_1$ be a deepest leaf of the tree. Let $X_i$ denote the prefix $x_j x_{j-1} \ldots x_i$ of $X$. $X_i$ occurs at least $i$ times in $T$ because it has at least $i$ descendants, $\{X_i, X_{i-1}, \ldots, X_1\}$, and each of these contains an occurrence of a substring of which $X_i$ is a prefix. Therefore, $X_{\lceil j/2 \rceil}$ has length $\lfloor j/2 \rfloor$ and occurs at least $\lceil j/2 \rceil$ times in $T$. It must be that $\lfloor j/2 \rfloor$ is a lower bound on $h(T)$, so the height $j$ is bounded by $2h(T) + 1$. □

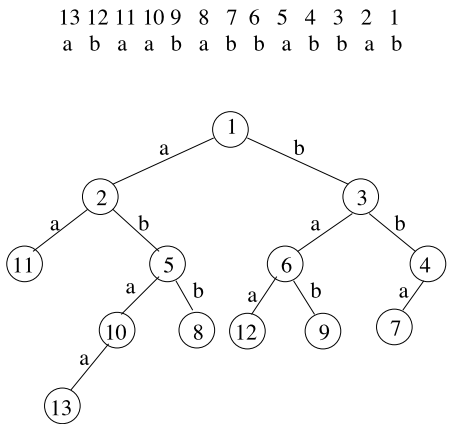


**Fig. 3.** The naive query algorithm. To find the occurrences of *ba*, index in on *ba* to the node labeled 6. All positions at descendants of this node ($\{6, 12, 9\}$) are occurrences of *ba*. In addition, some ancestors can be occurrences of *ba*. This is determined by inspection at the positions in ancestors, whereupon it is determined that 3 is also an occurrence. A string such as *babbabbab* that is not a node of the position heap is handled slightly differently. Index in on the longest prefix that is a node of the string, in this case *bab*. Only the positions $\{1, 3, 6, 9\}$ in ancestors of this node can be occurrences. Which ones are occurrences is determined by inspection at these positions, whereupon it is determined that 9 is the only occurrence.

**Corollary 4.3.** *The naive construction algorithm takes* $O(nh(T))$ *time.*

**Proof.** Indexing into the heap to find the parent of the new node to be inserted for position $i$ takes time that is bounded by the height of the heap, hence $O(h(T))$ time. Adding the new child takes $O(1)$ time. Summing this over all positions gives an $O(nh(T))$ bound. □

## 5. The naive query algorithm

We now give a time bound for querying the position heap. We improve the time bound below, at the expense of adding elements to the data structure.

**Definition 5.1.** The **naive query algorithm** for finding all occurrences of a pattern string $P$ in $T$ consists of the following steps.

- Index into the position heap to find the longest prefix $X$ of $P$ that is a node of $H(T)$. For each ancestor $X'$ of $X$ (including X), look up the position $i$ stored in $X'$. Position $i$ is an occurrence of $X'$. Determine whether this occurrence is followed by $P - X'$. If it is, report $i$ as an occurrence of $P$.
- If $X = P$, also report all positions stored at descendants of $X$.

Fig. 3 gives an example. This algorithms is also simple enough to be taught and programmed in undergraduate data structures classes.

**Lemma 5.2.** *The naive query algorithm is correct.*

**Proof.** A node $X'$ contains a position $i$ where $X'$ occurs in $T$.

If $X'$ is a prefix of $P$, then it is an ancestor of $X$, and $i$ may or may not be an occurrence of $P$ in $T$, depending on whether the occurrence of $X'$ at $i$ is followed by $P - X'$. The test for this condition returns $i$ if and only if it is an occurrence of $P$.

If $P$ is a prefix of $X'$, then $X = P$, and since all prefixes of $X'$ occur at position $i$, so does $P$. This is reported during the traversal of the subtree rooted at $P$.

If the longest common prefix $Y$ of $P$ and $X'$ is neither $P$ nor $X'$, then the occurrence of $Y$ at $i$ is followed by the first letter of $X' - Y$, which is not the first letter of $P - Y$. Therefore, $i$ is not an occurrence of $P$. The query does not report $i$ in this case. □

**Lemma 5.3.** *The naive query algorithm runs in* $O(\min(m^2, mh(T)) + k)$ *time.*

**Proof.** If $X$ is the longest prefix of $P$ that is a node of the heap, it takes $O(\|X\|)$ time to find $X$ by indexing into the heap on $P$. For each of the $\|X\| + 1$ ancestors of $X$, we must look up the position $i$ stored in the ancestor, and determine in $O(m)$ time whether $P$ occurs at position $i$. Since $\|X\| \leqslant m$, this gives an $O(m^2)$ bound for this step. Since $\|X\|$ is $O(h(T))$, this also gives an $O(mh(T))$ bound for this part.

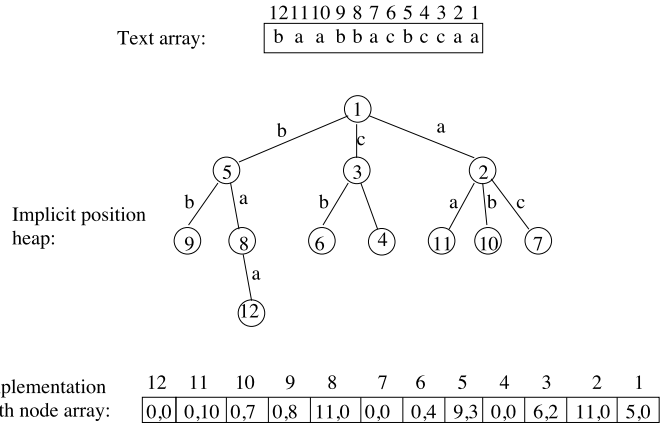


**Fig. 4.** A compact representation of the position heap. The data structure is an array of nodes, each of which has two integers. The first integer is the index of the left child, and the second is the index of the right sibling. The root is at position 1; an index of 0 indicates that a left child or right sibling does not exist. No pointer from a node to the text is needed, because the node at position $i$ points to text position $i$, and no labeling of edges is needed, because these can be calculated using arithmetic on the indices.

If $X = P$, that is, if $P$ is a node of the position heap, it also takes $O(1)$ time to return each of the positions in the subtree rooted at $X$, for a total of $O(m^2 + k)$ and $O(mh(T) + k)$. □

**Lemma 5.4.** *If $T$ is a randomly constructed string and the construction of $P$ does not depend on $T$, or if $P$ is a randomly constructed string and construction of $T$ does not depend on $P$, then the naive query algorithm takes $O(m + k)$ expected time, where $m$ and $k$ are as in Lemma 5.3.*

**Proof.** The $mh(T)$ term comes from the fact that at each of $O(h(T))$ nodes $X'$, we must check whether the occurrence of $X'$ at the position $i$ that it stores is followed by $P - X$. This requires checking whether $\|P - X\|$ letters of $P$ match at $\|P - X\|$ positions of $T$. The check halts when a mismatch is detected. The probability of any of positions matching is $1/|\Sigma|$, so the expected number of checks before halting is $(\Sigma - 1)/\Sigma \sum_{i=1}^{\|P-X\|} 1/|\Sigma|^i = O(1)$. □

### *5.1. A space-efficient representation for small alphabets*

We have omitted the size $|\Sigma|$ from our time bounds by assuming the alphabet size is fixed, and observing that there is a way to implement all algorithms so that the size of the alphabet contributes an $O(\log|\Sigma|)$ factor to the running times.

There is a way to implement the position heap so that it takes only $2n$ integers, at the expense of adding a $|\Sigma|$ factor to the time bounds. It is therefore suitable when space is at a premium and the alphabet size is not too large. The trick is to create an array of position-heap nodes, each of which consists of an ordered pair of integers. The first integer is the index of the left child, and the second is the index of the right sibling. An index of 0 indicates that there is no such node. Fig. 4 gives an example.

Given this, one may find the children of a node by going to the leftmost child and following right sibling links.

The text is also stored in an array, and the node for the character at position $i$ is stored at index $i$ of the node array. The root of the position heap is stored at index 1. When indexing into the heap, one always knows the index $i$ of the current node and the depth of the node. There is no need to store a pointer to the corresponding position of the text, as that occurs at position $i$ in its array. There is also no need for labels of edges from parent to child. If a node at index $i$ has depth $j$, then it is reached from its parent on the letter at $T[i - j + 1]$. Finding the child reachable on letter $c$ takes time proportional to the number of children, as all right-sibling links of the children need to be traversed in the worst case. The number of children is bounded by $|\Sigma|$, so finding this child takes $O(|\Sigma|)$ time.

## 6. The augmented position heap

The only obstacle to an $O(m + k)$ worst-case bound for returning the $k$ occurrences of $P$ is the time to check whether $P$ occurs at the positions stored at ancestors of the largest prefix $X$ of $P$ that is a node of the position heap.

**Definition 6.1.** Let $i$ be the position stored at node $X$ in $H(T)$, and let $Y$ be the largest prefix of $T_i$ that is a node of $H(T)$. The **maximal-reach pointer for** $X$ is a pointer from node $X$ to node $Y$. The **augmented position heap** for $T$ is obtained by labeling each node $X$ of $H(T)$ with its maximal-reach pointer and $X$'s discovery and finishing time in a depth-first traversal of $H(T)$ [4]. We also associate with the heap an array `N[]` such that `N[i]` contains a pointer to the node of the heap that contains position $i$. Let $H'(T)$ denote the augmented position heap.

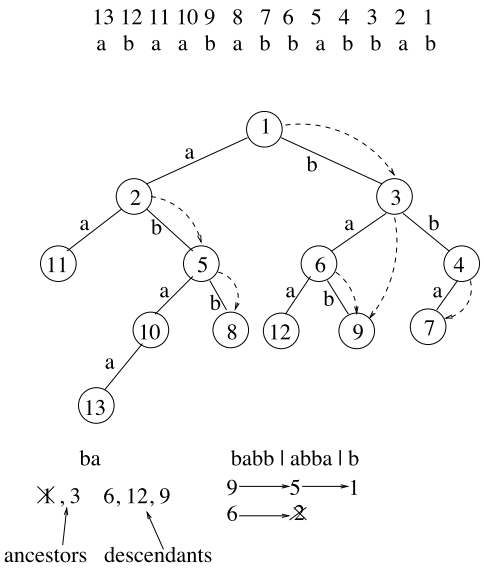


**Fig. 5.** The linear query algorithm on strings *ba* and *babbabbab* on the augmented position heap. Maximal-reach pointers are dashed, and maximal-reach pointers that are loops are omitted from the diagram.

Fig. 5 gives an example. The `N[]` array and the discovery and finishing times are omitted. The maximal-reach pointers are depicted with dashed arrows. For example, the maximal-reach pointer from the node *bb* labeled 4 points to the node *bba* labeled 7, since *bba* is the longest substring beginning at position 4 that is a node of the position heap.

A naive algorithm for obtaining the augmented position heap is obtained as follows.

> Create the position heap for $T$. The pointers can be installed in `N[]` and the discovery and finishing times can be assigned to nodes of the heap during a depth-first traversal of $T$. Then for each suffix $T_i$ of $T$, index as far as possible into $H(T)$ on $T_i$ to find the maximal node $Y$ that is a prefix of $T_i$. Install a maximal-reach pointer to $Y$ from the node $X$ pointed to by `N[i]`.

### 6.1. Queries in $O(m + k)$ time

It is well known that node $x$ of a rooted tree is an ancestor of node $y$ if and only if the discovery time of $x$ is less than the discovery time of $y$ and the finishing time of $x$ is larger than the finishing time of $y$ [4]. This gives the following:

**Lemma 6.2.** *Given pointers to two nodes $X$ and $Y$ of $H'(T)$, it takes $O(1)$ time to determine whether $X$ is an ancestor of $Y$.*

**Lemma 6.3.** *Given a pointer to a node $X$ of $H'(T)$ and a position $i$, it takes $O(1)$ time to determine whether $i$ is an occurrence of string $X$ in $T$.*

**Proof.** It takes $O(1)$ time to find the node $Y$ that contains $i$ using `N[]`. By Lemma 6.2, it takes $O(1)$ time to determine whether $Y$ is a descendant of $X$. If so, then since $i$ is an occurrence of $Y$ and $X$ is a prefix of $Y$, $i$ is an occurrence of $X$.

If not, it takes $O(1)$ time to determine whether $Y$ is an ancestor of $X$, by Lemma 6.2. If it is, then let $Z$ be the node pointed to by the maximal-reach pointer of $Y$. Position $i$ is an occurrence of $X$ if and only if $X$ is a prefix of $T_i$. $Z$ is the maximal prefix of $T_i$ that is a node of the heap. Therefore, $X$ occurs at position $i$ if and only if it is a (not-necessarily proper) prefix of $Z$, that is, if and only if $Z$ is a descendant of $X$. This takes $O(1)$ time to determine, by Lemma 6.2. □

For example, in Fig. 5, given a pointer to the node *ba* (the one labeled 6), we can tell that 12 is a descendant by looking in `N[12]` to find a pointer to its node *baa*, and using the discovery and finishing times of *ba* and *baa* to determine that *baa* is a descendant. Therefore, it is an occurrence. We can tell that 3 is an occurrence by looking in `N[3]` to find its node *b*, using the discovery/finishing times to find that it is an ancestor of node *ab*, using its maximal-reach pointer to find the node *bab*, and using the discovery/finishing times of *ba* and *bab* to determine that *bab* is a descendant of *ba*. We can tell that 1 is not an occurrence, because its maximal-reach pointer doesn't point to a descendant of *ba*.

**Corollary 6.4.** *Let $Xc$ be a string such that $X$ is a node of the tree and $Xc$ is not. Given a pointer to $X$ and a position $j$, it takes $O(1)$ time to determine whether $j$ is an occurrence of $Xc$.*

**Table 1**
The linear query algorithm for use with the augmented position heap.

- **Case 1:** *$P$ is a node of $H'(T)$.* This is detected by indexing into $H'(T)$ on $P$, and gives node $P$. For each proper ancestor $X'$ of $P$, look up the position $i$ stored at $X'$, and determine whether it is an occurrence of $P$. In addition, report all positions recorded in the subtree rooted at $P$.
- **Case 2:** *$P$ is not a node of $H'(T)$.*

```
// Find an initial set of candidate positions
Let CurrentSubstring be the shortest prefix of P that is not a node of H'(T)
Let I be the set of positions where CurrentSubstring occurs
// Invariants: FinishedPrefix + RemainingSuffix = P;
// I is the set of positions where FinishedPrefix occurs in T
FinishedPrefix = CurrentSubstring
RemainingSuffix = P − CurrentSubstring
while RemainingSuffix is not a node of H'(T)
    Let CurrentSubstring be the shortest prefix of RemainingSuffix that is not a node of H'(T)
    I := {j | j ∈ I and the occurrence of FinishedPrefix at j in T is followed by an occurrence of CurrentSubstring}
    RemainingSuffix = RemainingSuffix − CurrentSubstring
    FinishedPrefix = FinishedPrefix + CurrentSubstring
CurrentSubstring = RemainingSuffix
Let I := {j | j ∈ I and the occurrence of CurrentPrefix at i is followed by CurrentSubstring}
```

**Proof.** By Lemma 6.3, it takes $O(1)$ time to determine whether $j$ is an occurrence of $X$. If it is, then it is an occurrence of $Xc$ if $c$ occurs at position $j - \|X\|$, which takes $O(1)$ time to check when $T$ is stored in an array. □

Before giving pseudocode for the linear-time query algorithm, we illustrate the main ideas in Fig. 5. There are two cases: Case 1, where the search string is a node of the position heap, and Case 2, where it is not.

Case 1 is illustrated by *ba*, which is the node labeled 6. By Lemma 6.3, we can now check in $O(1)$ time apiece which of the positions $\{1, 3\}$ at proper ancestors are occurrences of *ba*. Only 3 is; its node is the only proper ancestor with a maximal-reach pointer into *ba*'s subtree. That is $O(m)$ time so far. In addition, we report the labels of descendants $\{6, 12, 9\}$ in $O(1)$ time apiece, as before, in $O(k)$ time, for a total of $O(m + k)$ time.

Case 2 is illustrated by *babbabbab*, which is not a node of the heap. Our strategy is to partition the string into segments *babb*, *abba*, and *b*, which can be handled efficiently by Corollary 6.4 and Lemma 6.3. We use the corollary to find the occurrences of *babb*, discard those that are not followed by *abba*. This gives the occurrences of *babbabba*. We then use the lemma to discard from these occurrences those that are not followed by *b*.

To apply the corollary, we want all the segments except the last to be of the form $Xc$, where $X$ is a node of the tree and $Xc$ is not. The first such segment is *babb*. This is our *current substring*. As in the naive query algorithm, only ancestors of $X = bab$ can be positions of *babb*. These are labeled $\{1, 3, 6, 9\}$. By Corollary 6.4, we can determine which are occurrences of the current substring *babb* in $O(1)$ time apiece, for a total of $O(\|Xc\|)$ time. This leaves positions $\{6, 9\}$. The string *babb* becomes the *finished prefix*, its positions $\{6, 9\}$ are known, and the rest of the query string, *abbab* is the *remaining suffix*.

We now look for the prefix of the remaining suffix *abbab* of the form $Xc$, where $X$ is a node of the heap and $Xc$ is not. This is *abba*. We want to find which occurrences of $Xc$ follow occurrences of the finished prefix *babb*. To do this, we subtract the length of the finished prefix from each of the positions of the finished prefix and determine in $O(1)$ time whether it is an occurrence of $Xc$, by Corollary 6.4. In the example, subtracting $\|babb\| = 4$ from 9 gives 5, and we determine that 5 is an occurrence of *abba*. Therefore, 9 is an occurrence of *babbabba*. Subtracting 4 from 6 gives 2, and we determine that 2 is not an occurrence of *abba*. Therefore, of the initial possible positions of the search string, $\{6, 9\}$, only 9 survives the test. The finished prefix is now *babbabba*, the positions where it occurs are known to be $\{9\}$, and the remaining suffix is *b*.

When the remaining suffix is short enough to be a node of the tree, let us denote it $Y$. (In the example, $Y = b$.) We subtract the length of the finished prefix from each of its occurrences ($\{9\}$ for this example), and check whether each of these positions ($\{1\}$ in this example) is an occurrence of $Y$, using Lemma 6.3. Since position 1 is an occurrence of $Y = b$, position 9 is an occurrence of the original search string.

Generalizing from these examples, we get the algorithm of Table 1.

**Lemma 6.5.** *The linear query algorithm is correct.*

**Proof.** For Case 1, the procedure is the same as the naive algorithm, except that at each ancestor $X'$ of $P$, we determine whether $i$ is an occurrence of $P$ in $O(1)$ time, instead of $O(\|P\|)$ time, using Lemma 6.3.

For Case 2, by induction on the number of times `FinishedPrefix` is assigned, $I$ is the set of positions where `FinishedPrefix` occurs in $T$. In the final line, $P = FinishedPrefix + RemainingSuffix$, and $I$ is assigned to be those positions of *FinishedPrefix*. After the final step, $FinishedPrefix = P$, hence $I$ is the set of positions in $T$ where $P$ occurs. □

**Lemma 6.6.** *The linear query algorithm can be implemented in $O(m + k)$ time using the augmented position heap.*

**Proof.** Case 1 differs from the naive approach only in that it uses Lemma 6.3 to determine which ancestors of $P$ contain the position of an occurrence of $P$, reducing each of these tests from $O(\|P\|)$ to $O(1)$. Since there are $\|P\|+1$ ancestors of $P$, this takes $O(\|P\|)$ time. As in the naive query algorithm, all other occurrences of $P$ are found in $O(1)$ time apiece during a traversal of the subtree rooted at $P$, for a total of $O(\|P\|+k)$ time.

For Case 2, let $(P_1, P_2, \ldots, P_l)$ be the values taken on by *CurrentSubstring*, and let $(I_1, I_2, \ldots, I_l)$ be the values taken on by $I$.

To find the $i$th value $P_i$ of *CurrentSubstring*, index as far as possible on *RemainingSuffix* into $H'(T)$, yielding node $X_i$, and let $b$ be the next character of *RemainingSuffix* following prefix $X$. $P_i = X_i b$. Over all iterations, this takes time proportional to $\sum_{i=1}^{l} \|P_i\| = O(\|P\|)$. For $2 \leqslant i \leqslant l$, and each $j \in I_{i-1}$, it takes $O(1)$ time to determine whether the instance of *FinishedPrefix* at position $j$ is followed by $X_i$; this is determined by finding whether $j - \|\mathit{FinishedPrefix}\|$ is an occurrence of $X_i$, using Lemma 6.3. It then takes $O(1)$ time to determine whether this occurrence of $X_i$ is followed by an occurrence of $b$ at position $j - \|\mathit{FinishedPrefix}\| - \|X_i\|$. This determines whether the occurrence of *FinishedPrefix* at position $j$ is followed by $X_i b = P_i$. Therefore, it takes $O(1)$ time to determine, for each element of $I_{i-1}$, whether it remains in $i$.

By the naive algorithm, each $P_i$ has $O(\|P_i\|)$ occurrences, because $P_i$ is not a node of the tree, hence its occurrences can only be recorded in ancestors (prefixes) of $X_i$. Therefore, $\|I_i\| = O(\|P_i\|)$. Determining $I_l$ therefore takes $O(\sum_{i=1}^{l-1} \|I_i\|) = O(\sum_{i=1}^{l-1} \|P_i\|) = O(\|P\|)$ time. □

### 6.2. Returning positions one-by-one in left-to-right order

It is sometimes claimed that the suffix array returns all $k$ occurrences of $P$ in $O(m + \log n)$ time, even though $k$ can be superlinear in this bound. The reason is that it gives a pointer to a list of the positions. This time bound captures the fact that if the user wants to examine the first $k'$ positions, this takes $O(k')$ rather than $\Theta(k)$ time. One way to view this is that it returns an iterator in $O(m + \log n)$ time that then takes $O(1)$ time per position to return the positions.

The position heap can be implemented to have this property also, using a depth-first search that maintains a stack of active calls that have not yet made a recursive call on their last child. One use of such an iterator, however, is to examine the *first* $k'$ positions *in left-to-right order*. This is a common operation in text editors, for example. This can be implemented in $O(\log k)$ worst-case time per element, due to the fact that the node labels have the heap property (Definition 2.1).

We illustrate how to produce an iterator that returns them in right-to-left order; left-to-right order can be obtained by building the augmented position heap for the reverse of the text. The positions of nodes on the indexing path $X$ take $O(1)$ time to check. If $P = X$, then the descendants of $X$ might also have to be returned in left-to-right order. Keep a priority queue on the topmost nodes of the subtree of $X$ whose positions have not yet been returned. Because the positions have the heap property (Definition 2.1), the minimum position is among these nodes. Initially the priority queue has $X$ in it. Each time a new position is asked for, the minimum index $i$ in the priority queue is returned, and the positions in the children of the node containing $i$ are inserted to the priority queue. Since $\Sigma = O(1)$ and the size of the subtree is $O(k)$, the size of the priority queue is $O(k)$, and extracting $i$ and inserting its children takes $O(\log k)$ time.

### 6.3. Space requirements

If one is willing to accept an implicit $|\Sigma|$ factor, rather than an implicit $\log|\Sigma|$ factor in the time bounds, an implementation similar to the one of Fig. 4 takes five integers per position; the additional three integers are the discovery and finishing times and the index of the node pointed to by the maximal-reach pointer. There are similar increases in the space requirements for other implementations.

The $m^2$ term of the $O(m^2+k)$ query time for the naive heap is overly pessimistic in most cases (it has an $O(m)$ expected value for random strings). The naive position heap is therefore likely to be preferable for most practical settings, where space is the scarce resource.

## 7. Building the position heap in $O(n)$ time

Each time a node is added to the position heap, its parent must be located so that it can be added as a child. The reason the above algorithm for constructing the position heap from the root does not take $O(n)$ time is that indexing from the root to find this parent at each iteration is not an $O(1)$ operation.

### 7.1. The strategy

Indexing into the heap from the root is not the only way to find the parent of the new node at step $i$. Let $X_{i-1}$ be the node added at step $i-1$, let the first letter of $T_i$ be $b$, that is, let $T_i = bT_{i-1}$, and let $X_i$ be the node added at step $i$. Since $X_{i-1}$ is a prefix of $T_{i-1}$, $X_i = bY$, where $Y$ is a prefix of $T_{i-1}$, hence a (not necessarily proper) ancestor of $X_{i-1}$. By Lemma 7.3, below, $\|X_i\| = \|bY\| \leqslant \|X_{i-1}\| + 1$. In other words, the depth of the added node can increase by at most one at each iteration. This suggests the idea building a separate structure, which we will call the *dual heap*, for finding the parent of $X_i = bY$, given $Y$. We may then search upward from $X_{i-1}$ in the position heap, instead of downward from the root, in

order to find $Y$. This is not an $O(1)$ operation, because we may have to search upward through a lot of ancestors of $X_{i-1}$ to find $Y$, but each ancestor that we traverse decreases the depth of the next node, $X_i = bY$ by 1. Since the depth can build back up at the rate of at most one per iteration, this will give an $O(1)$ amortized bound per iteration.

Since $Y$ can be much shorter than $X_{i-1}$, the upward search might have to proceed through a large number of nodes on the path from $X_{i-1}$ toward the root before $Y$ is reached. However, the new node at step $i$, $bY$ is then much shorter than the node, $X_{i-1}$, inserted at the previous iteration. The cost of the operation is proportional to the decrease in depth from one iteration to the next. What makes the approach more efficient than the above approach is that depth of the new node inserted at successive iterations can grow by at most 1 from one iteration to the next, by Lemma 7.3. This allows us to amortize occasional large costs incurred in iterations where the depth decreases by a large amount over many iterations where the depth slowly builds up again at the rate of one per iteration.

The argument is the same as that for a stack with a *multipop* operation described in the chapter *Amortized Analysis* in the textbook [4].

### 7.2. Implementation

The following lemma is the basis of the claim that the depth in the tree at which the algorithm works must build up again slowly if there is a sudden large and costly decrease in the depth.

**Lemma 7.1.** *If $P$ is not a node of $H(T)$, it has fewer than $\|P\|$ occurrences in $T$.*

**Proof.** Every suffix of $T$ that has $P$ as a prefix results in a new node of the tree that is either a proper prefix of $P$ or that has $P$ as a prefix. Since $P$ does not occur in the tree, it is not a prefix of any node in the tree. Therefore, the number of suffixes of $T$ that have $P$ as a prefix, hence the number of occurrences of $P$, is bounded by the number of proper prefixes of $P$. □

Let us say that a set of $\mathcal{S}$ of strings is **hereditary** if, whenever $X \in \mathcal{S}$, every substring of $X$ is also in $\mathcal{S}$.

**Lemma 7.2.** *The nodes of the position heap are a hereditary set of strings.*

For example, in the final tree, node *abaa* is labeled with position 13 of Fig. 5. Its substrings *aba*, *baa*, *ab*, *ba*, *aa*, *a*, *b*, and the empty string are all nodes of the position heap; they are labeled with positions 10, 12, 5, 6, 11, 2, 3, 1, respectively.

**Proof.** Let us show this by induction on the length of $T_i = t_i t_{i-1} \dots t_1$. The lemma is trivially true for $H(T_1)$, which has only one node, the empty string. Otherwise, we adopt as the induction hypothesis that the nodes of $H(T_{i-1})$ have the hereditary property. Since $H(T_i)$ differs from $H(T_{i-1})$ only by the addition of a node $X$, $H(T_i)$ can only fail to have the hereditary property if some proper substring of $X$ fails to be a node of $T_i$.

This can't be the case if $\|X\| < 2$, since $\lambda$ is a node of $H(T_i)$. Suppose $\|X\| \geqslant 2$. We can then write $X$ as $aX'b$. The parent of $aX'b$ is $aX'$, hence it is a node of $H(T_{i-1})$. Since $aX'$ is longer than $X'$, $X'$ is a node in $H(T_{i-2})$ by the induction hypothesis. Also, $X'b$ is a prefix of $T_{i-1}$, and since $X'$ is a node of $T_{i-2}$, $X'b$ is either added at step $i-1$ or is already a node of $T_{i-2}$. In either case, it is a node of $H(T_{i-1})$. We conclude that $aX'$ and $X'b$ are nodes of $T_{i-1}$. By the induction hypothesis, every substring of $aX'$ and $X'b$ is a node of $T_{i-1}$, hence of $T_i$, and these are every proper substring of the new node $X = aX'b$. □

This hereditary property is not shared by arbitrary instances of Coffman and Eve's data structure, as the node labeled 9 in Fig. 1 is the string *abba*, but its substring *bba* is not a node of the tree. It is not even true when the keys are the suffixes of a text $T$ when they are not inserted in ascending order of length. Fig. 6 gives an example.

**Lemma 7.3.** *For $1 < i \leqslant \|T\|$, if $X_{i-1}$ is the node inserted at step $i-1$ and $X_i$ is the node inserted at step $i$, then $\|X_i\| \leqslant \|X_{i-1}\| + 1$.*

**Proof.** Let $a$ denote the first letter of $T_i$. $X_{i-1}$ is the shortest prefix of $T_{i-1}$ that is not already a node of $H(T_{i-2})$ and $X_i$ is the shortest such prefix of $aT_{i-1} = T_i$. Let $b$ denote the last letter of $X_i$. Then $X_i$ can be written as $aYb$ for some string $Y$.

Suppose $\|X_i\| \geqslant \|X_{i-1}\| + 2$. Then $X_{i-1}$ is a *proper* prefix of $Yb$. Since $X_{i-1}$ is the longest prefix of $T_{i-1}$ that is not a node of $H(T_{i-2})$, $Yb$ is not a node of $H(T_i)$. By the hereditary property, $Yb$ is a node of $H(T_i)$, since it is a substring of $X_i$, which is a node of $H(T_i)$. The only new node added to $H(T_{i-1})$ to get $H(T_i)$ is $aYb$, so $Yb$ was already a node of $H(T_{i-1})$, a contradiction. □

To insert a node to the position heap, we must find the parent. Since inserting the node after the parent is found takes $O(1)$ time, the only obstacle to getting a linear time bound is repeated indexing into the position heap to find the parent of each node to be added. We must use an alternative method to find this parent.

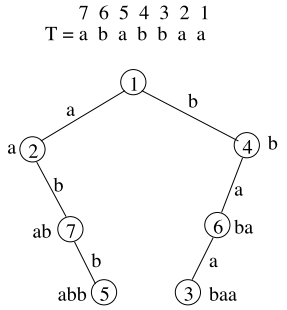


**Fig. 6.** The hereditary property doesn't necessarily apply when the suffixes are not inserted in order of ascending length. The figure depicts the Coffman and Eve structure where the insertion order of the suffixes is $(T_1, T_4, T_2, T_7, T_5, T_6, T_3)$. String *abb* is a node, but its substring *bb* is not a node of the tree.

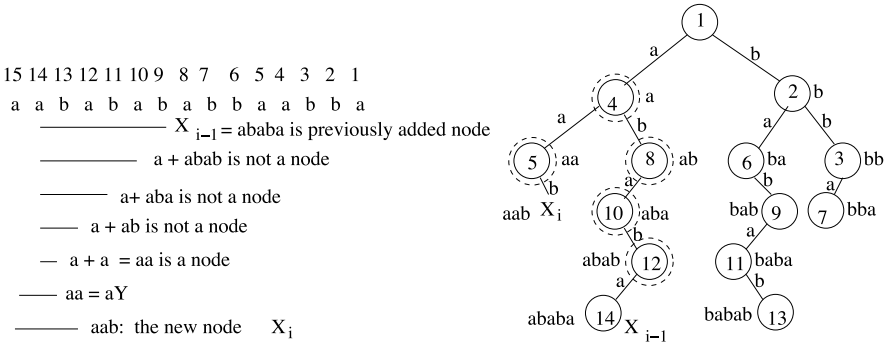


**Fig. 7.** Given the node $X_{i-1}$ added at step $i-1$, find the parent of the node $X_i$ added at step $i$.

The idea of our $O(n)$ alternative method is given in Fig. 7. At step $i-1=14$, we add $X_{i-1}=ababa$ as a new node. At step $i=15$, we must add the shortest prefix of $H(T_i)$ that is not already a node of the position heap. Let $a$ denote the first letter of $T_i$.

If the string $a$ does not already occur as a node of the position heap, then it can be added as a child of the root in $O(1)$ time.

Otherwise, as in the proof of Lemma 7.3, the new node must be $aYb$ for some prefix $Yb$ of the node $X_{i-1}$ added in step $i-1$, where $b$ is the character occurring $\|aY\|+1$ positions into $T_i$.

Below, we show how to find, for each such prefix $Y$ of $X_{i-1}$, whether $aY$ is already a node of the position heap, and if so, to return a pointer to it, in $O(1)$ time apiece. We try this on all proper prefixes of $X_{i-1}$ in descending order of length until we find the first. In the figure, we let $Y$ take on the sequence of values $(abab, aba, ab, a)$, whereupon it is discovered that $aY=aa$ is already a node of the position heap, and since the concatenation of $a$ and $ab$ is not, $aa$ is the longest prefix of $T_i$ that is already a node of the position heap. We have found the desired parent of the new node. The new node, $X_i=aYb$, is added as its child of $aY$ on an edge labeled with letter $b$.

This does not give an $O(1)$ bound to add each node of the tree. However, we can amortize the variable costs, showing that they sum to $O(n)$ over all iterations.

The reason the cost of step $i$ is not $O(1)$ is that we might have to try many prefixes $Y$ of $T_{i-1}$ before we find the one such that $aY$ is already a node of the heap. Let the *decrease in depth* denote the difference $\|X_i\|-\|X_{i-1}\|$ of the depth of the node added at position $i-1$ and the depth of the node added at position $i$. If this is negative, call it an *increase in depth*. If at step $i$, we try $k_i$ prefixes before finding $Y$ such that $aY$ is already a node of the tree, then we spent $O(k_i)$ time on the step, and $\|X_i\|=\|aYb\|=\|X_{i-1}\|-(k_i-2)$. The decrease in depth is $k_i-2$. The first two prefixes take $O(1)$ time, so the time spent at step $i$ is $O(1)$ plus the decrease in depth. By Lemma 7.3, the depth can increase by at most 1 at each iteration, so the total increase in depth is $O(n)$ over all iterations. The total decrease in depth can't exceed the total increase in depth, which means that over all iterations, the total decrease in depth is $O(n)$. Therefore, the total time spent by the algorithm is $nO(1)+O(n)=O(n)$.

It remains to describe how to get an $O(1)$ bound for finding, for each prefix $Y$ of $X_{i-1}$, whether $aY$ is already a node of the heap.

**Definition 7.4.** Let the **dual** $D(T)$ of the position heap $H(T)$ be the trie where for each node $X$ of $H(T)$, the **reverse** $X^R$ of $X$ is a node of $D(T)$ (see Fig. 8).

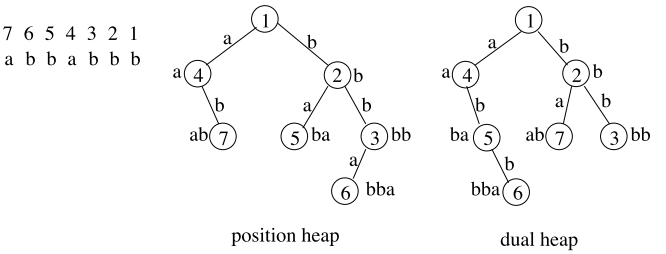


**Fig. 8.** The position heap and its dual for the text *abbabbb*. The labels of the path leading to a node in the dual is the reverse of the labels of the path leading to it in the position heap.

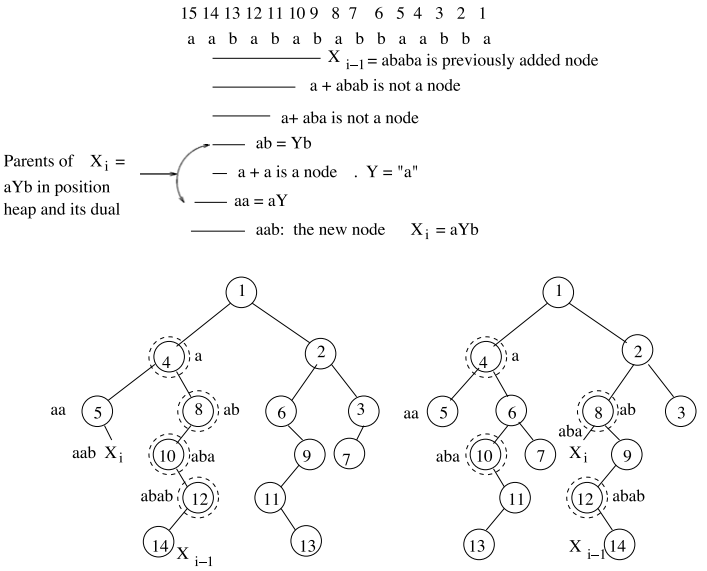


**Fig. 9.** Implementing the algorithm of Fig. 7 using the position heap and its dual. Starting at the previously-added node $X_{i-1}$, we find the lowest ancestor $Y$ such that $aY$ is already a node. This is accomplished by traversing ancestors in the position heap, and seeing if they have a child on edge labeled $a$ in the dual. In this case $Y$ is the node labeled 4. Its child on edge labeled $a$ in the dual is $aY$, the node labeled 5. It is the parent of the new node $X_i = aab$ in the position heap. The last prefix $ab$ tried before $Y$ was found is the longest node of the dual heap that is a prefix of $X$ and has no child labeled $a$. It is the parent of $X_i$ in the dual.

We continue to refer to each node by its path label $X$ in the position heap, even when considering it as a node of the dual. Equivalently, each node of $D(T)$ is denoted by the sequence $X$ of labels on edges from the node to the root of $D(T)$.

It is tempting to think that the dual is just the position heap of the reverse of the text, but it is easily verified that this is not the case.

**Lemma 7.5.** *The set of nodes of $D(T)$ is the same as the set of nodes of $H(T)$.*

**Proof.** Because for every node $X$ of $H(T)$, there is a node $X$ in $D(T)$, where $X$ is the string of labels from the node to the root in $D(T)$, every node of $H(T)$ is a node of $D(T)$. It remains to show that every node of $D(T)$ is a node of $H(T)$. Let $X$ be an arbitrary node of $H(T)$. By Lemma 7.2, not only is every prefix of a node $X$ of $H(T)$ a node of $H(T)$, but so is every suffix. This implies that every ancestor of $X$ in $D(T)$ is a node of $H(T)$. There are no nodes on any path of $D(T)$ that fail to be a node of $H(T)$. □

We implement the position heap and its dual on the same set of nodes, so that each node has both a parent in the position heap and a parent in the dual.

We concurrently construct the position heap and its dual. Suppose that at step $i$ we already have $H(T_{i-1})$ and $D(T_{i-1})$. We show how to update both to get $H(T_i)$ and $D(T_i)$ in $O(k_i)$ time.

When going from $H(T_{i-1})$ to $H(T_i)$, let $a$ be the first letter of $T_i$ and $X_{i-1}$ the node added at step $i-1$. (Refer to Fig. 9.) The prefixes of $Y$ in descending order of length are the ancestors encountered on the path from $X_{i-1}$ to the root of the position heap. For each such ancestor $Y$, we can find whether $aY$ is already a node of the heap by determining whether $Y$ has a child on an edge labeled $a$ in the dual. This takes $O(1)$ time, since $Y$ is both a node of the heap and of the dual.

We stop when we encounter the first one. By the above algorithm, this takes care of adding node $aXb$ to $H(T_{i-1})$, yielding $H(T_i)$ in $O(k_i)$ time.

However, we must also add this node to the dual, which requires locating its parent, $Xb$, and adding it as a child on edge labeled $a$. Fortunately, $Xb$ was just the last prefix of $Y$ considered before $X$ was discovered. We already found $Xb$ in the position heap, and since it is also a node of the dual, we have it in the dual. $aXb$ can be added as a child of $Xb$ on edge labeled $a$ in $O(1)$ additional time over what we have accounted for in adding it in the position heap. This gives the following:

**Lemma 7.6.** *It takes linear time to construct the position heap of a text $T$.*

### *7.3. Space requirements*

If one is willing to accept an implicit $|\Sigma|$ factor, rather than an implicit $\log|\Sigma|$ factor in the time bounds, then during construction the position heap and the dual heap can be structured as follows. The dual heap can be represented with the scheme of Fig. 4, while the position heap can be represented with an array of $n$ integers, where the element at index $i$ contains the index of the parent of node $i$. This is because the $O(n)$ algorithm for constructing the position heap always climbs upward in the position heap, and only traverses from parent to child in the dual heap.

Once the position heap is created, the dual heap can be discarded, and the space re-used to store the position heap using the structure of Fig. 4. Converting the representation where each node points to its parent to the representation of Fig. 4 in linear time is a trivial exercise.

Adding a new child to a node takes $O(1)$ time: the child's right sibling becomes the old leftmost child, and the new child becomes the new leftmost child. Constructing this array is a matter of allocating it and then filling it in, node-by-node, as prescribed by the above construction algorithm.

## 8. Constructing the augmented position heap in $O(n)$ time

The augmented position heap differs from the position heap in that the nodes are labeled with depth-first discovery and finishing times and with maximal-reach pointers. Depth-first search on a tree with $n$ nodes takes $O(n)$ time, so it only remains to describe how to compute the maximal-reach pointers in $O(n)$ time.

Once again, the strategy is to amortize the cost. The approach is virtually the same as it is for adding new nodes: instead of searching downward from the root at each iteration, we search upward in the tree, starting at the node pointed to by a maximal-reach pointer at the previous iteration. Even though this is not an $O(1)$ operation, the cost is proportional to the decrease in depth of the node pointed to by the maximal-reach pointer. This depth can increase by at most 1 from one iteration to the next, allowing to amortize large decreases in depth over many small increases in depth.

**Lemma 8.1.** *For $1 < i \leqslant \|T\|$, if $X_{i-1}$ is the node pointed to by the maximal-reach pointer of node $i-1$ and $X_i$ by the maximal-reach pointer of node $i$, then $\|X_i\| \leqslant \|X_{i-1}\| + 1$.*

**Proof.** Let $a$ denote the first letter of $T_i$. $X_{i-1}$ is the longest prefix of $T_{i-1}$ that is a node of $H(T)$, and $X_i$ is the longest prefix of $aT_{i-1} = T_i$ that is a node of $H(T)$. Let $b$ denote the last letter of $X_i$. Then $X_i$ can be written as $aYb$ for some string $Y$.

Suppose $\|X_i\| \geqslant \|X_{i-1}\| + 2$. Then $X_{i-1}$ is a *proper* prefix of $Yb$ and $Yb$ is not a node of $H(T_{i-1})$. By the hereditary property, $Yb$ is a node of $H(T_i)$, since it is a substring of $X_i$, which is a node of $H(T_i)$. The only new node added to $H(T_{i-1})$ to get $H(T_i)$ is $aYb$, so $Yb$ was already a node of $H(T_{i-1})$, a contradiction. □

To construct the *augmented* position heap in $O(n)$ time, our strategy is first to construct the position heap in $O(n)$ time using the algorithm from the previous section. As before, we create the array `N[]`, where `N[i]` points to the node that contains position $i$, and this takes $O(n)$ time by trivial methods. We then add the discovery and finishing times and the maximal-reach pointers on a second pass, in $O(n)$ time.

We find and test each prefix by starting at $X_{i-1}$ in the position heap and ascending through ancestors until we find the first one, $Y$, that has a child on edge labeled $a$ in the dual heap. This child, $aY$, in the dual is the node to which node $i$ must point.

The analysis of the linear running time is the same as it is for linear-time construction of the position heap. The *current depth* is the depth of node $X_i$ in the position heap. The first two prefixes of $X_{i-1}$ take $O(1)$ time to check for a child on edge labeled $a$ in the dual heap. Each additional prefix takes $O(1)$ time to check, and decreases the current depth in the position heap. Call this the *variable part* of the time spent at position $i$. By Lemma 8.1, the current depth can increase by at most one per iteration. The initial depth is at most 1, since $T_1$ has length 1. The total decrease in depth can therefore be at most one greater than the total increase in depth, which is $O(1)$ per iteration, hence $O(n)$ overall. The sum of the variable parts of the times spent at the different iterations is therefore $O(n)$. We therefore get the following:

**Lemma 8.2.** *It takes $O(n)$ time to construct the augmented position heap for a text $T$ of length $n$.*

## 9. Updating the position heap when the text is edited

When a block of characters is inserted to or deleted from a text $T$, the position heap must pass through a series of steps in which it is a trie, but has some things wrong with it that must be repaired in order for it to be the position heap of the new text. The goal of this section is to give algorithms for `Delete` and `Insert`, which update the position heap when a block of text is deleted from or inserted to the text $T$.

Since the text is no longer static, it is no longer convenient to label a node of the position heap with its position *number* in the text; when a position is deleted, the position numbers of all letters to its left decrease by one. To avoid having to update the position-number labels of all those nodes, we instead label the nodes with *position pointers* to the positions of the text. This requires us to define the analog of the heap property when pointers, rather than integers, are used.

**Definition 9.1.** If $p$ is a pointer to a position in $T$, let $T_p$ denote the suffix of $T$ that begins at $p$. If $X$ is a node in the trie with a pointer to a position of $T$, let $p(X)$ denote this pointer. The trie has the **heap property** if whenever $Y$ is a child of $X$, $p(Y)$ is to the left of $p(X)$ in $T$. The pointer $p(X)$ is **correctly placed** if $X$ has an occurrence at position $p(X)$, that is, if $X$ is a prefix of $T_{p(X)}$.

The constructive definition of the position heap (4.1) remains unchanged, except that each time a position is inserted, the new node is labeled with a pointer to the position, rather than its position number.

It will be convenient to look up the corresponding position-heap node given a pointer to a position in the text $T$. This is accomplished by labeling each position $p$ of the text with a pointer $N(p)$ to the node of the position heap that points to it. This serves the same function as the array $N[\,]$ in the static case. To avoid the need to mention this pointer each time we move a pointer in the position heap, we will define the operation of moving a position $p$ from one node to another in the position heap as including the operation of making the pointer $N(p)$ point to the new node.

The following lemma is useful for establishing that a procedure for updating the position heap after an edit operation on $T$ has correctly produced the position heap for the modified text.

**Lemma 9.2.** *A trie $H$ where each node is labeled with a pointer to a letter of a text $T$ is the position heap for $T$ if and only if it satisfies the following properties*:

1. *$H$ has the heap property*;
2. *Every position of $T$ is pointed to by **at most** one pointer $p(X)$ for some node $X$ in the trie*;
3. *Every position of $T$ is pointed to by **at least one** pointer $p(X)$ for some node $X$ of the trie*;
4. *For every node $X$, $p(X)$ is correctly placed.*

**Proof.** By induction on the number of positions inserted by the naive construction algorithm. □

### 9.1. Deleting or inserting a block of text in T

The workhorses of the algorithm for updating the position heap after insertion or removal of a block of text are `Remove` and `Add`. Below, we explain how they work, but for now, we define the problems in terms of their preconditions and postconditions so that, for the time being, we can make calls to them in our implementation of `Delete` and `Insert`.

**Definition 9.3** *(The problems solved by `Remove` and `Add`).* An input to `Remove` or `Add` is a trie that satisfies properties 1 and 2 of Lemma 9.2, but might not satisfy properties 3 and 4.

- An additional input to `Remove` is a node $X$ that contains a position pointer to be removed from the set of position pointers in the trie. It removes the pointer without disrupting the heap property, without otherwise changing the set of position pointers in the tree, **and without creating any new violations of property 4 at any position pointers.**
- An additional input to `Add` is a position pointer to be inserted to the trie. The position pointer must not already occur in the trie. It correctly places the pointer to without disrupting the heap property, without otherwise changing the set of position pointers in the tree, **and without creating any new violations of property 4 at any position pointers.**

A call to `Remove` or `Add` must update a variable $h$ that gives the current height of the trie.

Implementation requires shuffling position pointers in the tree in a way that is familiar to anyone who has studied heaps. Details are given below. In the meantime, given the problems solved by `Remove` and `Add`, we can now explain the main procedures of the section, `Delete` and `Insert`, in terms of calls to `Remove` and `Add`.

The `Delete` procedure updates the position heap when a block of characters is deleted from the text so that it is the position heap of the new text.

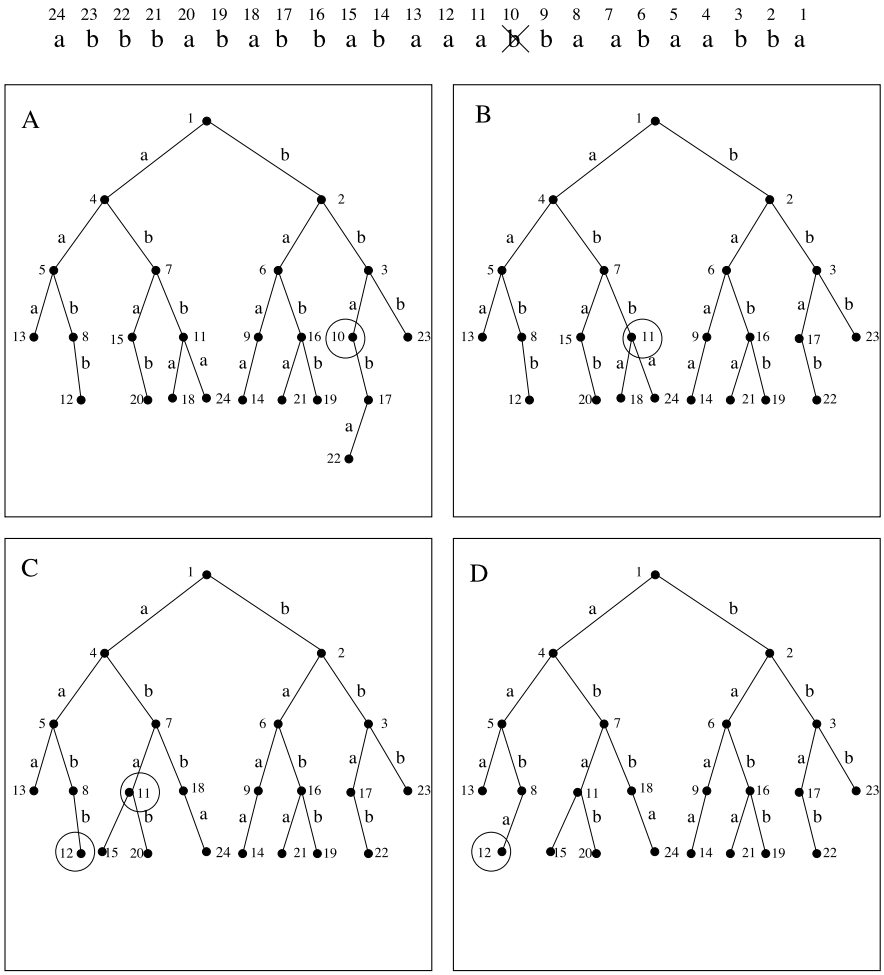


**Fig. 10.** Deletion of the *b* at position 10. First, position 10 is removed from the trie with `Remove` (figure B). Since position 11 resides at node *abb*, position 11 is supposed to be an occurrence of *abb*. This is no longer the case, because the deletion of position 10 removes the first *b* from this occurrence. Position 11 is no longer correctly placed, and this is fixed with a call to `Remove`, followed by a call to `Add` that correctly places it using the new string (figure C). Similarly, position 12 is no longer correctly placed and this must be repaired in the same way (figure D). If no node is a string that is longer than $h$, there can be no more than $h-1$ positions to the left of the edited position that are affected in this way.

**Definition 9.4** *(An algorithm for* `Delete`*).*

- Let $h$ be the height of the input position heap.
- Call `Remove` and `Add`, using the modified text, on the $h-1$ characters that lie to the left of the deleted block.
- Using `Remove`, remove the position pointers to the deleted characters.

The reason for the second step is illustrated in Fig. 10, where, for ease of understanding, the positions are identified with their position numbers in the original text, rather than with position pointers. The figure depicts the situation that arises when `Delete` is performed on just a single character, the one at position 10.

Each position $p$ that is correctly placed is stored at a node $X$ that has an occurrence at position $p$ in the original text. Position 11 is not involved in the edit, but it is no longer correctly placed because it is stored at *abb*. The occurrence of *abb* previously at position 11 no longer occurs, because the first *b* of it has been deleted at position 10. Therefore, position 11 is no longer correctly placed. Therefore, `Delete` calls `Remove` and `Add` on position 11 so that it is correctly placed, without creating any new incorrect placements. All such unedited positions that become incorrectly placed as a result of an edit lie within $h-1$ positions to the left of the edited position, because $h-1$, being the height of the tree, is the maximum length of the string that must occur at a position in order for it to be correctly placed. Let us call these $h-1$ positions the **affected positions**; they are not edited by the edit operation, but their placement in the heap is nevertheless affected by the edit.[1]

[1] We have defined `Delete` so that it performs a `Remove` and `Add` on positions on the four positions 11 through 14, since the height of the tree is 5. However, it is unnecessary to perform this operation on position 13. It is easy to see by Lemma 7.3, that once such a correctly-placed position is found to

**Lemma 9.5.** *`Delete` is correct.*

**Proof.** Initially, no pointer to the right of the edited position is incorrectly placed, since the naive algorithm inserts these in the same way, whether it is operating on the unedited or edited text. As explained above, an incorrectly placed pointer to the left of the edited positions must be among the $h-1$ affected positions. The pointers to the block of positions that have been removed are also incorrect.

Each call to `Remove` followed by `Add` correctly places an affected position pointer $p$ without changing the set of position pointers, disrupting the heap property, or creating new violations of property 4 at any other position pointers. The net effect of this is therefore to reduce the number of incorrectly placed position pointers by one, while maintaining the other properties. There are at most $h-1$ affected positions immediately to the left of the edited position, so when the second step has finished, there are no incorrectly placed position pointers to the left of the edited positions.

Because `Remove` removes the position pointers to the block of $b$ defunct positions, does not otherwise change the set of position pointers, and does not cause any new position pointers to be incorrectly placed. It follows that after this step, the trie adheres to all four properties of Lemma 9.2, so the modified trie is the position heap of the modified text. □

The `Insert` procedure updates the position heap when a character is inserted to the text.

**Definition 9.6** *(Implementation of an algorithm for `Insert`).*

- Using `Add`, insert position pointers to the $b$ new characters into the trie.
- Let $h$ be the current height of the trie.
- Call `Remove` and `Add` on the $h-1$ characters that lie to the left of the inserted block.

The proof of correctness of `Insert` is identical to that of `Delete`; it is essentially the inverses of the sequence of calls to `Add` and `Remove` in `Delete`.

### 9.2. Algorithms for `Remove` and `Add`

We now give algorithms for the `Remove` and `Add` operations, which remove or add a single position pointer to one of the intermediate tries during a call to `Delete` or `Add`. The `Remove` and `Add` operations are illustrated in Figs. 11 and 12.

**Definition 9.7** *(`Remove` on position $p$).* The position $p$ must be removed from the node $X$ that contains it in the current trie. Though a pointer to $X$ is given by $N(p)$, it is convenient to find the path from the root to $X$ by indexing into $T_p$, where $T$ is the text before the edit operation. This text is known from the new text and the edited block, which are both known by the call to `Delete` or `Insert` that calls `Remove`. This gives the depth of $X$ if $X$ is not the root.

Removing $p$ leaves $X$ with an empty position pointer. This must be filled without violating the heap property. This can be accomplished by finding the child $Y$ of $X$ whose position pointer into $T$ is rightmost, and promoting (moving) that position pointer to the parent. This, in turn, leaves an empty position pointer at $Y$, which may be filled recursively, ending at a base case where the node is a leaf, which is deleted.

To update the record $h$ of the height of the trie, we assume that `Remove` and `Add` jointly maintain a list that gives, for each depth, a count of the number of nodes at that depth. The counter at the depth of the removed leaf is decremented, and if this counter goes to 0, it is removed from the end of this list and the variable $h$ is decremented.

**Definition 9.8** *(`Add` position $p$).* If $p$ is the position pointer to be inserted, we index on $T_p$ until we can't index any farther, or else a position $q$ to the left of $p$ is found.

If we cannot index any farther, let $X$ be the last node on the indexing path. We create a new child $Y$ of $X$ reachable on letter $\|X\|+1$ of $T_p$ and store $p$ in it.

Otherwise $q$ must be **pushed down** to preserve the heap property. Let $X$ be the node that contains $q$. The push-down operation is accomplished as follows. As a base case, if $X$ has no child $Y$ reachable on character $\|X\|+1$ of $T_q$, then such a child is created and $q$ is stored in it. If $Y$ exists, the position pointer $r$ in $Y$ is pushed down recursively, and $q$ is stored in $Y$.

To update the record $h$ of the height, increment the counter for the number of nodes at the depth of the new leaf, and increment $h$ if the new leaf is the first node at that depth.

---

the left of the edited position, all positions to the left of the position are correctly placed. This observation can make the algorithm run somewhat faster, but since it does not affect the asymptotic bound, we omit the proof of it here.

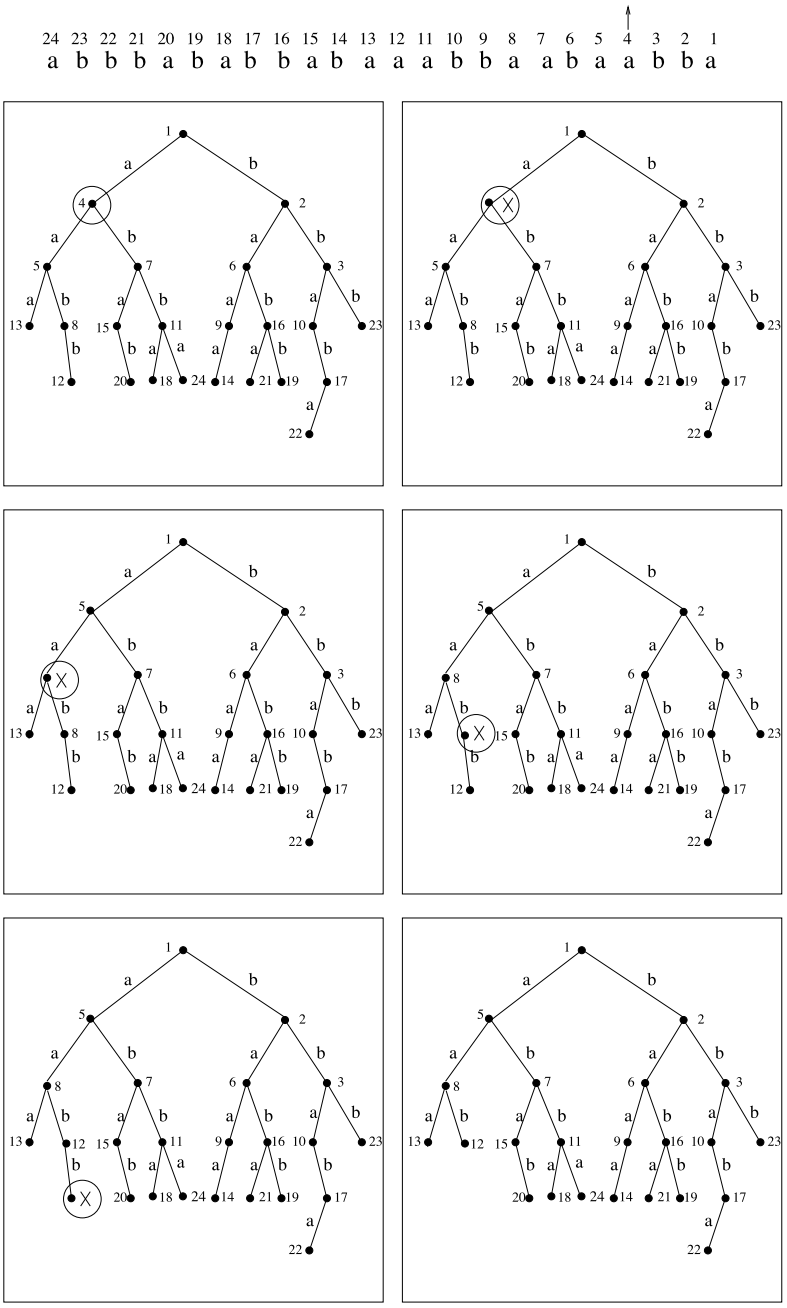

**Fig. 11.** Using the `Remove` operation to remove the pointer to position 4 from a position heap. Removing the pointer from its position-heap node leaves the node with an empty position pointer. This is filled by promoting the position pointer of the child whose position in $T$ is smallest (rightmost), in this case the pointer to position 5, to the empty parent. The child is now empty, and it is filled recursively. As a base case, the empty node is a leaf, and it is deleted. The only change to the shape of the tree is the deletion of a leaf.

### *9.3. Use of splay trees for representing dynamic texts and other lists*

When $T$ changes dynamically, we can no longer assume that the characters of $T$ are in an array. For completeness, we specify a straightforward way to represent a dynamic text in a way that supports array-like operations with a small extra cost. This allows us to generalize what we have developed above without defining too many new operations.

In particular, we use what we could call a `dynamic array abstract data type`, which supports the following operations on ordered lists.

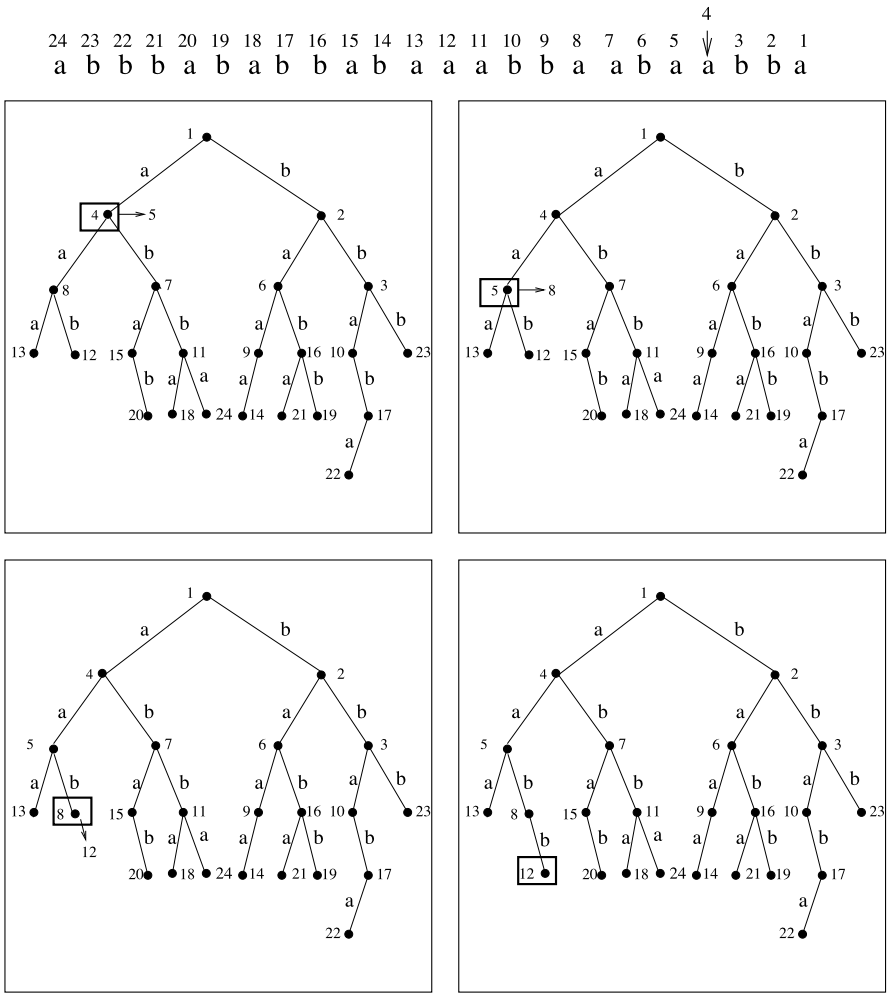


**Fig. 12.** The `Add` operation performs the inverse of `Remove`. To add the pointer to position 4 back in, index into the tree on $T_4 = abba$ until a node $X$ is encountered that has a larger position label. In this case, $X$ is the node with a pointer to position 5 in it. This pointer is pushed down to the child reachable on the letter $\|X\| + 1$ of $T_5$. Since $\|X\| = 1$, this is the letter $a$ at position 2 of $T_5 = aaba$. This makes it necessary to push down the pointer to position 8 recursively. As a base case, the pointer (to position 12) is pushed down to a new leaf. The only change to the shape of the tree is the addition of a new leaf.

- *makelist*$(v)$: Create a new dynamic array of length 1 containing a single element, $v$;
- *join*$(L_1, L_2)$: Join dynamic arrays $L_1$ and $L_2$, returning the concatenation $L_1L_2$;
- *split*$(L, v)$: Given element $v$ in dynamic array $L$, split $L$ into two dynamic arrays, $L_1$ consisting of the part of the array up through $v$, and $L_2$ consisting of the part of the array following $v$;
- $L[i]$: Return the element currently in position $i$ of $L$;
- $i(L, v)$: Given a pointer to element $v$ of $L$, return its current position number $i$ in $L$;

Sleator and Tarjan show how to implement `makelist` in $O(1)$ time, and *join* and *split* in $O(\log n)$ amortized time [12]. The data structure is based on *splay trees*, which are binary trees whose represented list is given by their inorder traversal. To obtain the bounds, every time they traverse a path in the tree, they perform a *splay* on the path. This consists of a set of rotations that alter the shape of the tree without altering the represented list. Their operations traverse the path from $v$ to the root, without affecting the $O(\log n)$ amortized bound.

The textbook [4] shows how to implement the last two operations of the abstract data type in time proportional to the path from the root to $v$ in their chapter, *Augmenting Data Structures*. The idea is to label each node $x$ with $size(x)$, which is the number of nodes in subtree rooted at $x$. Let $l$ and $r$ be the left and right children of the root. To look up $L[i]$, if $i \leqslant size(l)$, recursively find $L[i]$ in the left subtree, if $i = size(l) + 1$, then $L[i]$ is at the root, and if $i > size(l) + 1$, recursively find $L[i - size(l) - 1]$ in the right subtree. As always, the path traversed by this operation is splayed. During a rotation, it is easy to update the size labels of the nodes involved in the rotation in $O(1)$ time, using the size labels of their new children after the rotation.

This gives the following:

**Lemma 9.9.** *On the dynamic array abstract data type, the* `makelist` *operation takes $O(1)$ time, and each of the other operations takes $O(\log n)$ amortized time.*

**Corollary 9.10.** *Given two elements $v$ and $w$ of an instance $L$ of the dynamic array abstract data type, and another instance $L_2$ of the dynamic array abstract data type, the following take $O(\log n)$ amortized time*:

- *Determine which of $v$ and $w$ is earlier in $L$ by looking up $i(L, v)$ and $i(L, w)$;*
- *Remove the segment of $L$ between $v$ and $w$, yielding two dynamic arrays, what remains of $L$ and a dynamic array consisting of the removed section, by splitting $L$ before $v$ and after $w$, and joining the two dynamic arrays on either side of this section;*
- *Insert $L_2$ in $L$ starting at position $i$, by splitting $L$ before position $i$ and concatenating the three dynamic arrays.*

The data structure for the dynamic version of the position heap is identical to that of the static version, except that the dynamic array data structure is used to represent the text, rather than an array of characters, and instead of a position number, a node of the position heap has a pointer to the node of the dynamic array data structure. The position in memory of a splay-tree node never changes, even though the shape of the splay tree that represents it may, so the position pointers from the position heap do not have to be updated during a splay operation on a path. Above, we say that there must be a pointer from each position $p$ of the text to the position-heap node that contains $p$. This pointer can be stored as the label $N(p)$ on the corresponding splay-tree node in the representation of $T$.

To obtain amortized bounds, we assume that the text $T$ starts out as the empty string, and evolves through a series of insertions and deletions. The time bound is amortized beginning at this starting point.

### *9.4. A time bound for the naive query algorithm on the dynamic position heap*

In this subsection, we examine the effect of the new implementation on the time bound for the naive query algorithm given by Definition 5.1.

- Index into the position heap to find the longest prefix $X$ of $P$ that is a node of $H(T)$. For each ancestor $X'$ of $X$ (including $X$), look up the pointer $p$ into $T$ stored in $X'$. This is a pointer to an occurrence of $X'$ in $T$. Determine whether this occurrence is followed by $P - X'$ by calling $T[i(p) + \|X'\|], T[i(p) + \|X'\| + 1], \ldots, T[i(p) + \|X\| - 1]$ to find the substring of $T$ that follows this instance of $X'$, and comparing this with $P - X'$.
- If $X = P$, also report all positions stored at descendants of $X$.

Recall from Lemma 5.3 that, in the static case, the position heap takes $O(\min(m^2, mh(T)) + k)$ time.

**Lemma 9.11.** *The naive query algorithm on the dynamic position heap takes $O(\min(m^2, mh(T)) \log n + k)$ amortized time, where $m$ is the length of the query string, and $k$ is the number of occurrences of it in $T$.*

**Proof.** For each ancestor $X'$ of $X$, it takes $O(\|X\| \log n)$ amortized time to traverse the first $\|X\|$ characters of $T_{p(X)}$, comparing them with $X$. Since the height of the tree is $O(h(T))$, there are $O(\min(m, h(T)))$ ancestors of $X$. Multiplying the number of ancestors by the time spent at each ancestor gives $O(\min(m^2, mh(T)) \log n)$ time for these steps.

If $X = P$, that is, if $P$ is a node of the position heap, it also takes $O(k')$ time to return the $k'$ pointers in the subtree rooted in its subtree, each of which points to an occurrence of $P$. There are $O(k)$ of these, so they take $O(k)$ time to return. □

### *9.5. Time bounds for* `Delete` *and* `Insert`

**Lemma 9.12.** *The time for a call to* `Add` *or* `Remove` *on the dynamic implementation of a trie of height $h$ is $O(h \log n)$, amortized.*

**Proof.** For `Remove`, we must descend along a recursive path, identifying at each node the child that can be promoted without violating the heap property. This is the child whose pointer into the text $T$ is rightmost in $T$. Comparing two pointers to see which is rightmost requires $O(\log n)$ amortized time by Corollary 9.10, and it takes at most $|\Sigma| - 1$ such comparisons, which is $O(1)$ comparisons, since we have assumed that $\Sigma$ is $O(1)$. Promoting a pointer $p$ takes $O(1)$ time to move it to the parent, and $O(1)$ time to change the pointer $N(p)$ at position $p$ of the text so that it points to the new node where $p$ resides. The height of the tree is $O(h)$, so the total time is $O(h \log n)$, amortized. It takes $O(h)$ time to update the counts of the number of nodes at each level, and $O(1)$ to update the pointer $N(p)$ to the node that contains $p$.

For `Add`, we must find the node where the new position $p$ must be added. This requires indexing in on $T_p$ until we cannot index further, or else a node is found that contains a pointer $q$ that lies to the left of $p$ in $T$. Indexing requires looking up each successive character of $T_p$. This takes $O(\log n)$ amortized time per character. At each node $X$ on the

indexing path, we must determine whether the position $q$ at the node lies to the left of $p$, which takes $O(\log n)$ amortized time by Corollary 9.10. Since the height of the tree is $O(h)$, finding the node where $p$ must be added takes $O(h \log n)$ amortized time.

We must now analyze the time for the push-down operation. At each node $X$, we know $\|X\|$ because we found it by starting at the root, and each child is one character longer than its parent. We look up the pointer $q$ in $X$ in $O(1)$ time, find the character $\|X\|+1$ in $T_q$ in $O(\log n)$ amortized time by using the $L[]$ operator on the dynamic array implementation of $T$, and find the child of $X$ reachable on that character in $O(1)$ time. Moving a pointer $p$ from parent to child and updating $N(p)$ to reflect the move takes $O(1)$ time. The total time for push-down is $O(\log n)$, amortized, at each position on the push-down path, which has length $O(h)$. □

**Lemma 9.13.** *`Delete` on a block of $b$ characters takes $O((h(T)+b)h(T)\log n)$ amortized time, where $T$ is the text before it is edited.*

**Proof.** We perform a `Remove` and an `Add` on $h-1$ positions to the left of the edited block, where $h = O(h(T))$ is the initial height of the tree. Since `Remove` does not increase the height of the tree, and `Add` increases it by at most 1, the height can grow by at most $h-1$ during these operations, so it remains $O(h(T))$ throughout these steps. By Lemma 9.12, these $h-1$ operations take $O(h(T)^2 \log n)$ amortized time.

To remove the defunct positions in `Delete` takes $b$ calls to `Remove`. Since a call to `Remove` never increases the height of the tree, the tree has height $O(h(T))$ throughout this step. The $b$ calls to `Remove` on these positions takes $O(bh(T)\log n)$ amortized time. □

**Lemma 9.14.** *`Insert` on a block of $b$ characters takes $O((h(T')+b)h(T')\log n)$ amortized time, where $T'$ is the text after it is edited.*

**Proof.** Suppose a block of $b$ characters of a text $T'$ is removed, yielding text $T$. (We have switched the roles of $T$ and $T'$.) A call to `Delete` on text $T'$, resulting in the position heap of $T$, takes $O((h(T')+b)h(T')\log n)$ time by Lemma 9.13. This operation can be inverted by reinserting the block of $b$ deleted positions and calling `Insert`, yielding the position heap for $T$. This replaces the $b$ calls to `Remove` with $b$ calls to `Add`, which may be performed in reverse order, thereby stepping through the same sequence of tries in reverse order. Each call to `Remove` in `Delete` takes the same asymptotic bound as the inverse call to `Add`, so these operations take $O((h(T')+b)h(T')\log n)$, just as the call to `Delete` does.

As shown in the proof of the time bound for `Delete` the height $h$ of the tree after the calls to `Remove` and `Add` on the affected positions is $O(h(T'))$, so the final $h-1$ calls to `Remove` and `Add` take $O(h(T')^2 \log n)$ amortized time. □

## 10. A dynamic implementation of the augmented position heap

Recall that the additional features of the augmented position heap are a labeling of the nodes with maximal-reach pointers and discovery and finishing times.

### *10.1. Discovery and finishing times*

In the static case, the discovery and finishing times are integer labels from 1 to $2n$, where no two discovery/finishing labels are equal. The only purpose of these is so that we can compare two discovery times or two finishing times to find which is earlier; this is used in the test of whether one node is an ancestor of another. To support this operation on a dynamic structure, we consider discovery and finishing of nodes to be **events**, and create an **event list** using the dynamic array abstract data type described above. Instead of being labeled with integers to represent the discovery and finishing time, a node is labeled with two pointers into this list, one to its discovery event and one to its finishing event. This provides all of the functionality of the discovery- and finishing-time labels of the (static) augmented position heap, but at a cost of $\Theta(\log n)$ amortized time per comparison, rather than $O(1)$.

We must update the event list when the topology of the tree changes. Recall that the only effect of `Remove` or `Add` on the topology of the trie is the removal or addition of a leaf. If the leaf is removed, we remove its discovery and finishing events from the event list in $O(\log n)$ amortized time, using the leaf's pointers to these two events. If a new leaf $Z$ is added to the tree, we must find where its discovery and finishing events must be inserted to the event list. Since $Z$ is a leaf, its discovery and finishing events must be consecutive in the event list. If $Z$ is leftmost among its siblings, they are added immediately following the discovery event of its parent. Similarly, if the new leaf is rightmost among its siblings, they are added immediately preceding the finishing event of its parent. Otherwise, they are added immediately following the finishing event of the sibling to the left. The correctness of this placement follows from the order in which nodes are visited in a depth-first search. When $Z$ is inserted, `Add` has found the path from the root to $Z$, so the parent is known. It therefore takes $O(1)$ time to find the parent or left sibling of $Z$, and then $O(\log n)$ amortized time to insert the new events.

### 10.2. `Remove` and `Add` on the augmented dynamic position heap

Let us say that a trie is an **augmented trie** for a text if it satisfies properties 1 and 2 of Lemma 9.2 and its discovery- and finishing-event list is correct. It might not satisfy properties 3 and 4, and some of its maximal-reach pointers might not be correct.

If $X$ is a node of an augmented trie, let us denote its maximal-reach pointer by $m(X)$. The position pointer $p(X)$ of $X$ is the position pointer **corresponding** to $m(X)$ and *vice versa*. Recall that $m(X)$ is supposed to point to the node of the trie that is the maximal prefix of $T_{p(X)}$ that is a node of the trie. Let us say that $m(X)$ is **correct** if it satisfies this property.

A corresponding position pointer and maximal-reach pointer must reside at the same node, so when a position pointer is promoted (moved from child to parent), or pushed down (moved from parent to child), so must the corresponding maximal-reach pointer. When a position pointer is removed from the trie, so must the corresponding maximal-reach pointer. The asymptotic cost of performing these operations on the maximal-reach pointer is subsumed by the cost of performing them on the position pointer. When a call to `Add` inserts a new position $p$ to a node $X$, it must insert the maximal-reach pointer to $X$. The node that it must point to is found by indexing as far as possible into the trie on $T_{p(X)}$; it must point to the last node on this path. This takes $O(h \log n)$ amortized time, where $h$ is the current height of the trie. This is subsumed by the $O(h \log n)$ amortized cost of `Add`, as described above, for the unaugmented dynamic position heap.

A new issue arises in managing the maximal-reach pointers during a call to `Remove` that does not arise in managing position pointers. A call to `Remove` changes the topology of the trie by removing exactly one leaf $Z$. Let $P$ be the parent of $Z$. Let $q$ and $m$ be corresponding position and maximal-reach pointers, respectively. If $m$ points to $Z$, then $Z$ ceases to be the maximal prefix of $T_q$ that is a node of the trie; the next smaller prefix, $P$, is now this node. Since $Z$ is a descendant of the node occupied by $m$, this issue only affects maximal-reach pointers at ancestors of $Z$. Since `Remove` finds the path from the root to $Z$, we add a step that checks each node $Y$ on this path to see if $m(Y)$ points to $Z$, and, if so, changes it to point to $P$. This requires traversing $O(h)$ nodes, and at each, performing an $O(1)$ operation. This $O(h)$ cost is subsumed by the cost of `Remove`, as described above, for the unaugmented heap.

The issue also affects a call to `Add`, which changes the topology of the trie by adding $Z$ as a child of $P$. Let $c$ be the letter that labels the edge from $P$ to $Z$. If $m(X)$ initially points to $P$, then $P$ is a prefix of $T_{p(X)}$. If $Z = Pc$ is also a prefix of $T_{p(X)}$, then $m(X)$ must point to $Z$, not $P$, in order to be correct in the new trie. Since $P$ is a prefix of $T_{p(X)}$, this only happens at ancestors of $P$. We add a step to `Add` that traverses each node $X$ on the path from the root to $P$, determines whether $m(X)$ points to $P$, and, if so, whether the character $\|P\|$ positions to the right of $p(X)$ is $c$. If it satisfies all of these conditions, then $m(X)$ is changed to point to $Z$. Looking up the character $\|P\|$ positions to the right of $p$ takes $O(\log n)$ amortized time using the dynamic array implementation of the text, for a total of $O(h \log n)$ amortized time, which is subsumed by the cost of `Add` given above. Finally, $m(Z)$ is set to point to $Z$.

Let us call the operations of `Add` and `Remove` that carry these additional steps `Add2` and `Remove2`. Since the cost of all additional operations performed in `Add2` and `Remove2` are subsumed by the cost of operations in `Add` and `Remove`, we get the following:

**Lemma 10.1.** *The time for a call to `Add2` or `Remove2` on a trie of height $h$ is $O(h \log n)$, amortized.*

The additional operations in `Add2` and `Delete2` allow us to add the following preconditions and postconditions to them, in addition to the ones that apply to `Add` and `Remove` (Definition 9.3.)

1. For `Remove2`, the additional preconditions are that the event list is correct for input trie but that some maximal-reach pointers may be incorrect. The additional postconditions are that the event list is correct for the modified trie, that the maximal-reach pointer corresponding to the removed position has also been removed, *and that no other maximal-reach pointers have been made incorrect by the operation*.
2. The additional preconditions for `Add2` are also that the event list is correct for the input trie, and some maximal-reach pointers may be incorrect. The additional postconditions are that the event list is correct for the modified trie, that the maximal-reach pointer corresponding to the inserted position is correct, *and that no other maximal-reach pointers have been made incorrect by the operation*.

### 10.3. `Delete` and `Insert` on the augmented position heap

To modify `Delete` and `Insert` for the augmented position heap, we make them call `Remove2` and `Add2` in place of `Remove` and `Add`. Like the position pointers, the maximal-reach pointers corresponding to the $h-1$ positions to the left of the edited position can become incorrect, even though they correspond to positions that were not edited. The reason is exactly the same as it is for the position pointers. The maximal-reach pointer associated with a pointer $q$ is supposed to point to the longest node $Y$ that is a prefix of $T_q$. This node can be a string of length as long as $h$. Since $q$ is within $h-1$ positions of where $T$ is edited, $Y$ may no longer be a prefix of $T_q$ after the edit.

However, by the preconditions and postconditions of `Remove2` and `Add2`, calling `Remove2` followed by `Add2` on the $h-1$ positions preceding the edited position suffices to repair this problem at each of these positions without creating any new problems with position pointers or maximal-reach pointers at other nodes.

Since the asymptotic time bounds of `Remove2` and `Add2` are the same as those of `Remove` and `Add`, the analysis of the running time of `Delete` and `Insert` is unchanged when it operates on the augmented position heap. This gives the following, by Lemmas 9.13 and 9.14.

**Lemma 10.2.** *`Delete` on a block of $b$ characters takes $O((h(T)+b)h(T)\log n)$ amortized time on the augmented position heap, where $T$ is the text before it is edited.*

**Lemma 10.3.** *`Insert` on a block of $b$ characters takes $O((h(T')+b)h(T')\log n)$ amortized time, where $T'$ is the text after it is edited.*

*10.4. Time bound for queries on the dynamic augmented position heap*

**Lemma 10.4.** *Using the augmented dynamic position heap, it takes $O(m\log n+k)$ amortized time to find the $k$ occurrences of a string $P$ in $T$.*

**Proof.** This is obtained by reexamining the proof of Lemma 6.6 in light of the fact that the use of dynamic arrays causes some operations that previously took $O(1)$ time to take $O(\log n)$ amortized time. These are determining whether a node $Y$ is a descendant of a node $X$, which takes $O(\log n)$ amortized time: we look up whether $X$'s discovery event precedes $Y$'s and $X$'s finishing even follows $Y$'s in the dynamic-array implementation of the event list. This is accomplished with the $i()$ operator on dynamic arrays. This contrasts with the $O(1)$ time these comparisons take when the discovery and finishing times are implemented with integers. We must also determine whether a known occurrence of a string $X_i$ is followed by a letter $b$. The position of the occurrence and the length of $X_i$ gives the position of the next letter. Looking up this letter is accomplished with the $L[]$ operator on dynamic arrays, which takes $O(\log n)$ amortized time instead of the $O(1)$ time it takes on an array implementation of the text. There are $O(m)$ of these operations, for an $O(m\log n)$ amortized bound for them. Any remaining occurrences of the query string are reported by traversing the subtree rooted at the query string, which takes time proportional to the number of positions in this subtree. □

## References


[1] A. Blumer, J. Blumer, D. Ehrenfeucht, D. Haussler, R. McConnell, Complete inverted files for efficient text retrieval and analysis, Journal of the ACM 34 (1987) 578–595.
[2] M. Burrows, D. Wheeler, A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation.
[3] E. Coffman, J. Eve, File structures using hashing functions, Communications of the ACM 13 (1970) 427–432.
[4] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, McGraw–Hill, Boston, 2001.
[5] A. Ehrenfeucht, R.M. McConnell, String searching, in: D. Mehta, S. Sahni (Eds.), Handbook of Data Structures and Applications, CRC Press, 2005.
[6] A. Ehrenfeucht, R.M. McConnell, S.-W. Woo, Contracted suffix trees: a simple and dynamic text indexing data structure, Combinatorial Pattern Matching (2009) 41–53.
[7] P. Ferragina, R. Grossi, M. Montangero, On updating suffix tree labels, Theoretical Computer Science 201 (1–2) (1998) 249–262, http://dx.doi.org/10.1016/S0304-3975(97)00243-0.
[8] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proceedings of the 41st Annual Symposium on Foundations of Computer Science, vol. 41, 2000, pp. 390–490.
[9] U. Manber, E. Myers, Suffix arrays: a new method for on-line search, SIAM Journal on Computing 22 (1993) 935–948.
[10] M. Salson, T. Lecroq, M. Léonard, L. Mouchard, Dynamic extended suffix arrays, Journal of Discrete Algorithms 8 (2010) 241–257.
[11] M. Salson, T. Lecroq, M. Léonard, L. Mouchard, Dynamic Burrows–Wheeler transform, in: J. Holub, J. Žďárek (Eds.), Proceedings of the Prague Stringology Conference 2008, Czech Technical University in Prague, Czech Republic, ISBN 978-80-01-04145-1, 2008, pp. 13–25.
[12] R.E. Tarjan, Data Structures and Network Algorithms, Society for Industrial and Applied Math., Philadelphia, 1983.
[13] P. Weiner, Linear pattern-matching algorithms, in: Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory, Institute of Electrical Electronics Engineers, London, 1973, pp. 1–11.